# S&DS 685: Problem Set 2

**Name:** Ananya Krishna **NetID:** ark89 **Date:** 02/18/2025 **Collaborator:** Valentina Simon

## 1 Linear Markov Decision Processes (Pages 1-2)

**(a)** For any value function $V : S \to \mathbb{R}$, the transition operator is defined as

$$(PV)(s,a) = \mathbb{E}_{s' \sim P(\cdot|s,a)}\big[V(s')\big] = \sum_{s' \in S} P(s' \mid s,a)\, V(s').$$

We also know from equation 1 that

$$P(s' \mid s,a) = \phi(s,a)^\top \mu(s').$$

so we can substitute for $P(s' \mid s,a)$ to give

$$\begin{aligned}(PV)(s,a) &= \sum_{s' \in S} \Big[\phi(s,a)^\top \mu(s')\Big]\, V(s') \\ &= \phi(s,a)^\top \left(\sum_{s' \in S} \mu(s')\, V(s')\right).\end{aligned}$$

If we define

$$v_V := \sum_{s' \in S} \mu(s')\, V(s') \in \mathbb{R}^d,$$

then we obtain

$$(PV)(s,a) = \phi(s,a)^\top v_V.$$

This shows that the value $(PV)(s,a)$ is obtained by taking the dot product of $\phi(s,a)$ with a vector that is solely dependent on V and the model parameters. Hence, the operator $P$ applied to any value function $V$ yields a linear function of $\phi(s,a)$.

Now, we have a fixed policy $\pi$ with corresponding state-value function $V^\pi$ and action-value function $Q^\pi$. The Bellman evaluation equation for the action-value function is

$$Q^\pi(s,a) = R(s,a) + \gamma\, (PV^\pi)(s,a).$$

We can substitute the reward function from (2) and the result above:

$$\begin{aligned}Q^\pi(s,a) &= \phi(s,a)^\top \theta + \gamma\, \phi(s,a)^\top \left(\sum_{s' \in S} \mu(s')\, V^\pi(s')\right) \\ &= \phi(s,a)^\top \left(\theta + \gamma \sum_{s' \in S} \mu(s')\, V^\pi(s')\right).\end{aligned}$$

Defining

$$w^\pi := \theta + \gamma \sum_{s' \in S} \mu(s')\, V^\pi(s') \in \mathbb{R}^d,$$

we can express the action-value function as

$$\boxed{Q^\pi(s,a) = \phi(s,a)^\top w^\pi.} \tag{4}$$

This shows that the $Q$-function is linear in $\phi(s,a)$.

**(b)** In a tabular MDP, the dimension is $d = |S \times A|$. We proceed using equation 2.

Since $\phi(s,a)$ is a one-hot vector that is 1 in the coordinate corresponding to $(s,a)$ and 0 elsewhere, the inner product $\phi(s,a)^\top \theta$ simply extracts the $(s,a)$-th component of $\theta$. Hence, for the equality to hold for all $(s,a)$, we must have

$$\theta(s,a) = R(s,a) \quad \text{for all } (s,a) \in S \times A$$

Similarly, the linear MDP model posits that

$$P(s' \mid s,a) = \phi(s,a)^\top \mu(s'),$$

where $\mu(s') \in \mathbb{R}^{|S \times A|}$ is a vector associated with state $s'$. Since $\phi(s,a)$ is one-hot (with a 1 in the coordinate corresponding to $(s,a)$), we have

$$P(s' \mid s,a) = [\phi(s,a)](s,a) \cdot \mu(s')(s,a) = \mu(s')(s,a).$$

Thus, the parameter $\mu$ can be viewed as a matrix in $\mathbb{R}^{|S| \times |S \times A|}$ whose entry in row $s'$ and column $(s,a)$ is given by

$$\mu(s')(s,a) = P(s' \mid s,a).$$

In other words, the matrix $\mu$ encodes the transition probabilities exactly as in the tabular case.

## 2 Natural Policy Gradient and Policy Mirror Descent (Pages 3-7)

**(a)** The given PMD update is

$$\pi_{\theta_{\text{new}}}(\cdot \mid s) \propto \pi_\theta(\cdot \mid s) \cdot \exp\Big(\alpha\, Q^{\pi_\theta}(s, \cdot)\Big)$$

We also know that the advantage function is

$$A^{\pi_\theta}(s, a) = Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s).$$

so

$$Q^{\pi_\theta}(s, a) = A^{\pi_\theta}(s, a) + V^{\pi_\theta}(s).$$

Plugging this into the update gives

$$\pi_{\theta_{\text{new}}}(a \mid s) \propto \pi_\theta(a \mid s) \cdot \exp\Big(\alpha\, [A^{\pi_\theta}(s, a) + V^{\pi_\theta}(s)]\Big).$$

Since $V^{\pi_\theta}(s)$ does not depend on the action $a$, the factor $\exp(\alpha\, V^{\pi_\theta}(s))$ is the same for all $a$ and cancels out upon normalization. Therefore, the update is equivalent to

$$\pi_{\theta_{\text{new}}}(a \mid s) \propto \pi_\theta(a \mid s) \cdot \exp\Big(\alpha\, A^{\pi_\theta}(s, a)\Big).$$

Now, assume we update the parameters as

$$\theta_{\text{new}}(s, a) = \theta(s, a) + \alpha\, A^{\pi_\theta}(s, a), \quad \forall (s, a).$$

Since our softmax policy is defined by

$$\pi_\theta(a \mid s) = \frac{e^{\theta(s,a)}}{\sum_{a'} e^{\theta(s,a')}},$$

the updated policy becomes

$$\pi_{\theta_{\text{new}}}(a \mid s) = \frac{e^{\theta(s,a) + \alpha\, A^{\pi_\theta}(s,a)}}{\sum_{a'} e^{\theta(s,a') + \alpha\, A^{\pi_\theta}(s,a')}}.$$

$$\pi_{\theta_{\text{new}}}(a \mid s) = \frac{e^{\theta(s,a)} e^{\alpha\, A^{\pi_\theta}(s,a)}}{\sum_{a'} e^{\theta(s,a')} e^{\alpha\, A^{\pi_\theta}(s,a')}}.$$

Noting that

$$\pi_\theta(a \mid s) = \frac{e^{\theta(s,a)}}{\sum_{a'} e^{\theta(s,a')}},$$

we can express the updated policy as

$$\pi_{\theta_{\text{new}}}(a \mid s) = \pi_\theta(a \mid s) \cdot \frac{e^{\alpha\, A^{\pi_\theta}(s,a)}}{\sum_{a'} \pi_\theta(a' \mid s) e^{\alpha\, A^{\pi_\theta}(s,a')}} = \pi_\theta(a \mid s) \cdot \exp\Big(\alpha\, A^{\pi_\theta}(s, a)\Big).$$

This is exactly the same as the PMD update derived earlier. Hence, the PMD update is equivalent to the parameter update

$$\theta_{\text{new}}(s,a) = \theta(s,a) + \alpha\, A^{\pi_\theta}(s,a), \quad \forall (s,a).$$

**(b)** For a fixed state $s$ the softmax policy is

$$\pi_\theta(a \mid s) = \frac{e^{\theta(s,a)}}{\sum_{a'} e^{\theta(s,a')}}.$$

Taking the logarithm, we have

$$\log \pi_\theta(a \mid s) = \theta(s,a) - \log\Big(\sum_{a'} e^{\theta(s,a')}\Big).$$

Differentiate with respect to $\theta(s,a')$:

Case 1: $a' = a$.
The derivative of $\theta(s,a)$ with respect to $\theta(s,a)$ is 1. The derivative of $-\log\Big(\sum_{a''} e^{\theta(s,a'')}\Big)$ with respect to $\theta(s,a)$ is

$$-\frac{1}{\sum_{a''} e^{\theta(s,a'')}} e^{\theta(s,a)} = -\pi_\theta(a \mid s).$$

Thus,

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta(s,a)} = 1 - \pi_\theta(a \mid s).$$

Case 2: $a' \neq a$.
The derivative of $\theta(s,a)$ with respect to $\theta(s,a')$ is 0. The derivative of $-\log\Big(\sum_{a''} e^{\theta(s,a'')}\Big)$ with respect to $\theta(s,a')$ is

$$-\frac{1}{\sum_{a''} e^{\theta(s,a'')}} e^{\theta(s,a')} = -\pi_\theta(a' \mid s).$$

Thus,

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta(s,a')} = -\pi_\theta(a' \mid s).$$

Combining these two cases, we have

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta(s,a')} = \mathbf{1}\{a' = a\} - \pi_\theta(a' \mid s).$$

Moreover, if $s' \neq s$, then $\theta(s', a')$ does not appear in $\log \pi_\theta(a \mid s)$, and so

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta(s', a')} = 0.$$

**(c)** We wish to compute the gradient of the expected return

$$J(\pi_\theta) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t\right]$$

with respect to $\theta(s, a)$. The policy gradient theorem states that

$$\nabla J(\pi_\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s)\, \nabla \log \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s, a),$$

where $d^{\pi_\theta}(s)$ is the discounted visitation distribution. Focusing on the component corresponding to $\theta(s, a)$, we have

$$\frac{\partial J(\pi_\theta)}{\partial \theta(s, a)} = d^{\pi_\theta}(s) \sum_{a'} \pi_\theta(a' \mid s) \frac{\partial \log \pi_\theta(a' \mid s)}{\partial \theta(s, a)} Q^{\pi_\theta}(s, a').$$

Using the result from part (b),

$$\frac{\partial \log \pi_\theta(a' \mid s)}{\partial \theta(s, a)} = \mathbf{1}\{a' = a\} - \pi_\theta(a \mid s),$$

we obtain

$$\frac{\partial J(\pi_\theta)}{\partial \theta(s, a)} = d^{\pi_\theta}(s) \left[\pi_\theta(a \mid s)\, Q^{\pi_\theta}(s, a) - \pi_\theta(a \mid s) \sum_{a'} \pi_\theta(a' \mid s)\, Q^{\pi_\theta}(s, a')\right].$$

Noting that

$$\sum_{a'} \pi_\theta(a' \mid s)\, Q^{\pi_\theta}(s, a') = V^{\pi_\theta}(s),$$

we simplify the expression to

$$\frac{\partial J(\pi_\theta)}{\partial \theta(s, a)} = d^{\pi_\theta}(s)\, \pi_\theta(a \mid s) \left[Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s)\right] = d^{\pi_\theta}(s)\, \pi_\theta(a \mid s)\, A^{\pi_\theta}(s, a).$$

**(d)** For any state $s$, we define

$$F_s(\theta) = \mathbb{E}_{a \sim \pi_\theta(\cdot \mid s)}\left[\nabla_\theta \log \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s)^\top\right],$$

where here $\theta$ is viewed as the vector $\{\theta(s,a) : a \in A\}$ corresponding to state $s$.

From part (b), we have for any $a, a' \in A$,

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta(s,a')} = \mathbf{1}\{a' = a\} - \pi_\theta(a' \mid s).$$

Define the vector $g(a)$ whose $a'$-th component is

$$g(a)_{a'} = \mathbf{1}\{a' = a\} - \pi_\theta(a' \mid s).$$

Then,

$$F_s(\theta) = \sum_{a \in A} \pi_\theta(a \mid s)\, g(a)\, g(a)^\top.$$

For a categorical distribution, the covariance matrix is given by

$$\mathrm{Cov}(x) = \mathrm{diag}\big(\pi_\theta(\cdot \mid s)\big) - \pi_\theta(\cdot \mid s)\, \pi_\theta(\cdot \mid s)^\top.$$

Since $g(a) = e_a - \pi_\theta(\cdot \mid s)$ where $e_a$ is the one-hot vector for action $a$, we deduce that

$$F_s(\theta) = \mathrm{diag}\big(\pi_\theta(\cdot \mid s)\big) - \pi_\theta(\cdot \mid s)\, \pi_\theta(\cdot \mid s)^\top.$$

**(e)** Assume that $d^{\pi_\theta}(s) > 0$ for all $s$. For a fixed state $s$, let $A^{\pi_\theta}(s,\cdot) \in \mathbb{R}^{|A|}$ be the vector of advantages. We need to show that

$$F_s(\theta)\, A^{\pi_\theta}(s,\cdot) = \pi_\theta(\cdot \mid s) \odot A^{\pi_\theta}(s,\cdot),$$

where $\odot$ denotes element-wise multiplication.

From part (d)

$$F_s(\theta) = \mathrm{diag}\big(\pi_\theta(\cdot \mid s)\big) - \pi_\theta(\cdot \mid s)\, \pi_\theta(\cdot \mid s)^\top.$$

Let $v = A^{\pi_\theta}(s,\cdot)$ and denote by $\pi$ the vector $\pi_\theta(\cdot \mid s)$. Then, for any action $a$, the $a$-th component of $F_s(\theta)\, v$ is

$$\begin{aligned}\Big[F_s(\theta)\, v\Big]_a &= \pi(a \mid s)\, v_a - \pi(a \mid s) \sum_{a' \in A} \pi(a' \mid s)\, v_{a'} \\ &= \pi(a \mid s)\, \big[v_a - \mathbb{E}_{a' \sim \pi}\big[v_{a'}\big]\big].\end{aligned}$$

Since the advantage function satisfies

$$\mathbb{E}_{a' \sim \pi}[A^{\pi_\theta}(s,a')] = \sum_{a'} \pi(a' \mid s)\, \Big(Q^{\pi_\theta}(s,a') - V^{\pi_\theta}(s)\Big) = V^{\pi_\theta}(s) - V^{\pi_\theta}(s) = 0,$$

it follows that

$$\Big[F_s(\theta)\, v\Big]_a = \pi(a \mid s)\, v_a.$$

In vector form, this is exactly

$$F_s(\theta)\, A^{\pi_\theta}(s,\cdot) = \pi_\theta(\cdot \mid s) \odot A^{\pi_\theta}(s,\cdot).$$

This result implies that

$$A^{\pi_\theta}(s,\cdot) = F_s(\theta)^{-1}\Big[\pi_\theta(\cdot \mid s) \odot A^{\pi_\theta}(s,\cdot)\Big],$$

which shows that when the Fisher information matrix is used to precondition the gradient, we obtain the advantage function. In other words, the NPG update direction satisfies

$$\Delta_{\mathrm{NPG}} = F(\theta)^{-1}\nabla_\theta J(\pi_\theta) = A^{\pi_\theta},$$

which is equivalent to the PMD update in the tabular softmax case.

## 3 NPG & PMD - Softmax Linear Policy (Pages 7-10)

**(a)** The PMD update is given by

$$\pi_{\theta_{\text{new}}}(a \mid s) \propto \pi_\theta(a \mid s) \exp\Big\{\alpha\, Q^{\pi_\theta}(s,a)\Big\}, \quad \forall s \in S.$$

Since by assumption

$$Q^{\pi_\theta}(s,a) = \phi(s,a)^\top \eta^{\pi_\theta},$$

we have

$$\pi_{\theta_{\text{new}}}(a \mid s) \propto \pi_\theta(a \mid s)\ \exp\Big\{\alpha\, \phi(s,a)^\top \eta^{\pi_\theta}\Big\}.$$

Recall that the current policy is

$$\pi_\theta(a \mid s) = \frac{\exp\{\phi(s,a)^\top \theta\}}{Z_\theta(s)},$$

where $Z_\theta(s) = \sum_{a' \in A} \exp\{\phi(s,a')^\top \theta\}$. Then

$$\pi_{\theta_{\text{new}}}(a \mid s) \propto \exp\{\phi(s,a)^\top \theta\}\ \exp\Big\{\alpha\, \phi(s,a)^\top \eta^{\pi_\theta}\Big\} = \exp\Big\{\phi(s,a)^\top \big(\theta + \alpha\, \eta^{\pi_\theta}\big)\Big\}.$$

To obtain the proper probability distribution, we must normalize over all actions. Define the new partition function

$$Z_{\theta_{\text{new}}}(s) = \sum_{a \in A} \exp\Big\{\phi(s,a)^\top \theta + \alpha\, \phi(s,a)^\top \eta^{\pi_\theta}\Big\}.$$

Then, the normalized new policy is

$$\pi_{\theta_{\text{new}}}(a \mid s) = \frac{\exp\Big\{\phi(s,a)^\top \theta + \alpha\, \phi(s,a)^\top \eta^{\pi_\theta}\Big\}}{Z_{\theta_{\text{new}}}(s)} = \frac{\exp\Big\{\phi(s,a)^\top \big(\theta + \alpha\, \eta^{\pi_\theta}\big)\Big\}}{\sum_{a' \in A} \exp\Big\{\phi(s,a')^\top \big(\theta + \alpha\, \eta^{\pi_\theta}\big)\Big\}}.$$

This is exactly the softmax policy with parameter

$$\theta_{\text{new}} = \theta + \alpha\, \eta^{\pi_\theta}.$$

Thus, we conclude that the PMD update is equivalent to the parameter update.

**(b)** For the softmax-linear policy,

$$\pi_\theta(a \mid s) = \frac{\exp\{\phi(s,a)^\top \theta\}}{\sum_{a' \in A} \exp\{\phi(s,a')^\top \theta\}},$$

its logarithm is

$$\log \pi_\theta(a \mid s) = \phi(s,a)^\top \theta - \log\Big(\sum_{a' \in A} \exp\{\phi(s,a')^\top \theta\}\Big).$$

Taking the gradient with respect to $\theta$ (noting that the parameter $\theta$ is shared across actions at state $s$) gives

$$\nabla_\theta \log \pi_\theta(a \mid s) = \phi(s,a) - \frac{\sum_{a' \in A} \exp\{\phi(s,a')^\top \theta\}\, \phi(s,a')}{\sum_{a' \in A} \exp\{\phi(s,a')^\top \theta\}} = \phi(s,a) - \bar{\phi}(s),$$

where we define

$$\bar{\phi}(s) := \sum_{a' \in A} \pi_\theta(a' \mid s)\, \phi(s,a').$$

Moreover, if $s' \neq s$ then $\theta(s,a)$ and $\theta(s',a')$ are independent, so

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta(s',a')} = 0.$$

**(c)** The policy gradient theorem tells us that

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s \sim d_{\pi_\theta},\, a \sim \pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s,a)\Big].$$

Substituting the result from part (b) and using the linearity assumption $Q^{\pi_\theta}(s,a) = \phi(s,a)^\top \eta^{\pi_\theta}$, we have

$$\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{s \sim d_{\pi_\theta},\, a \sim \pi_\theta}\Big[\Big(\phi(s,a) - \bar{\phi}(s)\Big)\Big(\phi(s,a)^\top \eta^{\pi_\theta}\Big)\Big] \\
&= \mathbb{E}_{s \sim d_{\pi_\theta}}\left[\mathbb{E}_{a \sim \pi_\theta}\Big[\phi(s,a)\phi(s,a)^\top\Big]\eta^{\pi_\theta} - \bar{\phi}(s)\bar{\phi}(s)^\top \eta^{\pi_\theta}\right] \\
&= \mathbb{E}_{s \sim d_{\pi_\theta}}\Big[\big(\mathbb{E}_{a \sim \pi_\theta}\big[\phi(s,a)\phi(s,a)^\top\big] - \bar{\phi}(s)\bar{\phi}(s)^\top\big)\eta^{\pi_\theta}\Big].
\end{aligned}$$

Define, for each $s \in S$,

$$F_s(\theta) := \mathbb{E}_{a \sim \pi_\theta}\big[\phi(s,a)\phi(s,a)^\top\big] - \bar{\phi}(s)\bar{\phi}(s)^\top.$$

Then, if we denote

$$F(\theta) := \mathbb{E}_{s \sim d_{\pi_\theta}}\Big[F_s(\theta)\Big],$$

the policy gradient becomes

$$\nabla_\theta J(\pi_\theta) = F(\theta)\, \eta^{\pi_\theta}.$$

**(d)** The NPG update is given by

$$\theta_{\text{new}} = \theta + \alpha\, F(\theta)^{-1} \nabla_\theta J(\pi_\theta).$$

Using the result from part (c) (equation (10)), we have

$$\nabla_\theta J(\pi_\theta) = F(\theta)\, \eta^{\pi_\theta}.$$

Substitute this into the NPG update:

$$\theta_{\text{new}} = \theta + \alpha\, F(\theta)^{-1}\, F(\theta)\, \eta^{\pi_\theta} = \theta + \alpha\, \eta^{\pi_\theta}.$$

Thus, the NPG update reduces to the same parameter update as in part (a):

$$\boxed{\theta_{\text{new}} = \theta + \alpha\, \eta^{\pi_\theta}.}$$

# 4 Revisit NPG = PMG (Pages 11-12)

**(a)**

We wish to show that for a small perturbation $\Delta\theta$, the KL divergence between $\pi_\theta$ and $\pi_{\theta+\Delta\theta}$ satisfies

$$D_{\mathrm{KL}}\big(\pi_\theta \,\|\, \pi_{\theta+\Delta\theta}\big) \approx \frac{1}{2}\Delta\theta^\top I(\theta)\Delta\theta,$$

where the Fisher information matrix is

$$I(\theta) = \mathbb{E}_{s\sim d,\, a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s)\, \nabla_\theta \log \pi_\theta(a|s)^\top\Big].$$

Define the function

$$f(\theta') = D_{\mathrm{KL}}\big(\pi_\theta \,\|\, \pi_{\theta'}\big) = \mathbb{E}_{s\sim d}\, \mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\log \frac{\pi_\theta(a|s)}{\pi_{\theta'}(a|s)}\right].$$

By definition, when $\theta' = \theta$ we have $f(\theta) = 0$. A Taylor expansion of $f$ around $\theta' = \theta$ gives

$$f(\theta+\Delta\theta) = f(\theta) + \nabla_{\theta'} f(\theta)\Big|_{\theta'=\theta}^\top \Delta\theta + \frac{1}{2}\Delta\theta^\top \nabla^2_{\theta'\theta'} f(\theta)\Big|_{\theta'=\theta}\Delta\theta + o(\|\Delta\theta\|^2).$$

Since $f(\theta) = 0$ and the first-order term vanishes (because the KL divergence is minimized at $\theta' = \theta$), we have

$$D_{\mathrm{KL}}\big(\pi_\theta \,\|\, \pi_{\theta+\Delta\theta}\big) = f(\theta+\Delta\theta) \approx \frac{1}{2}\Delta\theta^\top \nabla^2_{\theta'\theta'} f(\theta)\Big|_{\theta'=\theta}\Delta\theta.$$

The Hessian of the KL divergence at $\theta' = \theta$ equals the Fisher information matrix. This is because

$$\mathbb{E}_{a\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a|s)\, \nabla_\theta \log \pi_\theta(a|s)^\top\Big] = -\mathbb{E}_{a\sim\pi_\theta}\left[\nabla^2_{\theta\theta} \log \pi_\theta(a|s)\right].$$

Thus,

$$\nabla^2_{\theta'\theta'} f(\theta)\Big|_{\theta'=\theta} = I(\theta).$$

It follows that

$$D_{\mathrm{KL}}\big(\pi_\theta \,\|\, \pi_{\theta+\Delta\theta}\big) \approx \frac{1}{2}\Delta\theta^\top I(\theta)\Delta\theta.$$

**(b)** Define the performance improvement term as

$$E \coloneqq \mathbb{E}_{s\sim d_{\pi_\theta}}\left[\langle Q^{\pi_\theta}(s,\cdot),\, \pi_{\theta'}(\cdot|s) - \pi_\theta(\cdot|s)\rangle\right].$$

We wish to show that for a small parameter change, i.e. for $\theta' = \theta + \Delta\theta$, we have

$$E \approx \Delta\theta^\top \nabla_\theta J(\pi_\theta).$$

Recall that the performance difference (by the policy gradient theorem) is given by

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim d_{\pi_\theta}} \mathbb{E}_{a\sim\pi_\theta(\cdot|s)} \left[ Q^{\pi_\theta}(s,a) \nabla_\theta \log \pi_\theta(a|s) \right].$$

Now, perform a first-order Taylor expansion of $\pi_{\theta'}(a|s)$ around $\theta$:

$$\pi_{\theta'}(a|s) \approx \pi_\theta(a|s) + \Delta\theta^\top \nabla_\theta \pi_\theta(a|s).$$

Hence, the difference in policies is approximately

$$\pi_{\theta'}(a|s) - \pi_\theta(a|s) \approx \Delta\theta^\top \nabla_\theta \pi_\theta(a|s).$$

Then, the inner product in the performance improvement term becomes

$$\langle Q^{\pi_\theta}(s,\cdot),\, \pi_{\theta'}(\cdot|s) - \pi_\theta(\cdot|s) \rangle \approx \sum_{a\in A} Q^{\pi_\theta}(s,a)\, \Delta\theta^\top \nabla_\theta \pi_\theta(a|s).$$

Interchange the sum and inner product to write:

$$\langle Q^{\pi_\theta}(s,\cdot),\, \pi_{\theta'}(\cdot|s) - \pi_\theta(\cdot|s) \rangle \approx \Delta\theta^\top \left( \sum_{a\in A} Q^{\pi_\theta}(s,a)\, \nabla_\theta \pi_\theta(a|s) \right).$$

Now, note that we can express

$$\nabla_\theta \pi_\theta(a|s) = \pi_\theta(a|s)\, \nabla_\theta \log \pi_\theta(a|s),$$

so that

$$\sum_{a\in A} Q^{\pi_\theta}(s,a)\, \nabla_\theta \pi_\theta(a|s) = \sum_{a\in A} \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a)\, \nabla_\theta \log \pi_\theta(a|s).$$

Taking the expectation over $s \sim d_{\pi_\theta}$ gives

$$E \approx \Delta\theta^\top\, \mathbb{E}_{s\sim d_{\pi_\theta}} \left[ \sum_{a\in A} \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a)\, \nabla_\theta \log \pi_\theta(a|s) \right].$$

By the policy gradient theorem, the term in square brackets is exactly $\nabla_\theta J(\pi_\theta)$. Hence,

$$E \approx \Delta\theta^\top \nabla_\theta J(\pi_\theta).$$

## 5 Analysis of Policy Mirror Descent (Pages 13 - 15)

**(a)** ]

$$V_1^{e\pi}(s_1) - V_1(s_1) = \sum_{h=1}^{H} \Big\{ \big[\mathcal{T}_h^{e\pi} V_{h+1}\big](s_h) - V_h(s_h) \Big\},$$

where $\mathcal{T}_h^{e\pi}$ is the Bellman operator at stage $h$ when actions are chosen according to $e\pi$.

Our goal is to relate $\Delta_1(s_1)$ to the per-stage differences in the policies and the error terms $e_h$.

By definition, the Bellman equation in this case is

$$V_h^{e\pi}(s) = \mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ R_h(s,a) + \mathbb{E}_{s'\sim P_h(\cdot|s,a)} \big[V_{h+1}^{e\pi}(s')\big] \Big].$$

Subtract $V_h(s) = \langle Q_h(s,\cdot),\, \pi(\cdot \mid s)\rangle$ from both sides to obtain

$$\Delta_h(s) = V_h^{e\pi}(s) - V_h(s) = \mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ R_h(s,a) + \mathbb{E}_{s'\sim P_h(\cdot|s,a)} \big[V_{h+1}^{e\pi}(s')\big] \Big] - V_h(s).$$

Now add and subtract $Q_h(s,a)$ inside the expectation to get

$$\Delta_h(s) = \mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ Q_h(s,a) + e_h(s,a) + \mathbb{E}_{s'\sim P_h(\cdot|s,a)} \big[V_{h+1}^{e\pi}(s')\big] \Big] - V_h(s).$$

We can split the expectation into two parts:

$$\Delta_h(s) = \underbrace{\mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ Q_h(s,a) \Big] - V_h(s)}_{(i)} + \underbrace{\mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ e_h(s,a) \Big]}_{(ii)} + \underbrace{\mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ \mathbb{E}_{s'\sim P_h(\cdot|s,a)} \big[V_{h+1}^{e\pi}(s')\big] \Big]}_{(iii)}.$$

For term (i), note that by the definition of $V_h(s)$ we have

$$V_h(s) = \langle Q_h(s,\cdot),\, \pi(\cdot \mid s)\rangle,$$

so that

$$\mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ Q_h(s,a) \Big] - V_h(s) = \langle Q_h(s,\cdot),\, e\pi_h(\cdot \mid s) - \pi(\cdot \mid s)\rangle.$$

Also, term (iii) can be written as

$$\mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ \mathbb{E}_{s'\sim P_h(\cdot|s,a)} \big[V_{h+1}^{e\pi}(s')\big] \Big] = \mathbb{E}_{s'\sim P_h^{e\pi}(\cdot|s)} \Big[ V_{h+1}^{e\pi}(s') \Big],$$

where $P_h^{e\pi}(\cdot \mid s)$ is the distribution over next states when following $e\pi_h$ at state $s$.

Thus, we have the recursion

$$\Delta_h(s) = \langle Q_h(s,\cdot),\, e\pi_h(\cdot \mid s) - \pi(\cdot \mid s)\rangle + \mathbb{E}_{a\sim e\pi_h(\cdot|s)} \Big[ e_h(s,a) \Big] + \mathbb{E}_{s'\sim P_h^{e\pi}(\cdot|s)} \Big[ \Delta_{h+1}(s') \Big].$$

Now we take the expectation over the state $s$ at stage $h$ under the distribution $\bar{d}_h^{e\pi}$ (the marginal of $s_h$ when following $e\pi$). For each $h$,

$$\mathbb{E}_{s\sim\bar{d}_h^{e\pi}}\Big[\Delta_h(s)\Big] = \mathbb{E}_{s\sim\bar{d}_h^{e\pi}}\Big[\langle Q_h(s,\cdot),\, e\pi_h(\cdot\mid s) - \pi(\cdot\mid s)\rangle\Big] + \mathbb{E}_{(s,a)\sim d_h^{e\pi}}\Big[e_h(s,a)\Big] + \mathbb{E}_{s\sim\bar{d}_h^{e\pi}}\Big[\mathbb{E}_{s'\sim P_h^{e\pi}(\cdot|s)}\big[\Delta_{h+1}(s')\big]\Big].$$

But by the law of total expectation, the last term equals the expected value of $\Delta_{h+1}(s')$ under the marginal $\bar{d}_{h+1}^{e\pi}$.

$$\mathbb{E}_{s\sim\bar{d}_h^{e\pi}}\Big[\mathbb{E}_{s'\sim P_h^{e\pi}(\cdot|s)}\big[\Delta_{h+1}(s')\big]\Big] = \mathbb{E}_{s'\sim\bar{d}_{h+1}^{e\pi}}\Big[\Delta_{h+1}(s')\Big].$$

We can sum the above equality over $h = 1,\dots,H$. $\Delta_{H+1}(s) = V_{H+1}^{e\pi}(s) - V_{H+1}(s) = 0$:

$$\mathbb{E}_{s_1\sim p_0}\Big[\Delta_1(s_1)\Big] = \sum_{h=1}^{H}\Big\{\mathbb{E}_{s\sim\bar{d}_h^{e\pi}}\Big[\langle Q_h(s,\cdot),\, e\pi_h(\cdot\mid s) - \pi(\cdot\mid s)\rangle\Big] + \mathbb{E}_{(s,a)\sim d_h^{e\pi}}\Big[e_h(s,a)\Big]\Big\}.$$

Recalling that $\Delta_1(s_1) = V_1^{e\pi}(s_1) - V_1(s_1)$, we have established that

$$\mathbb{E}_{s_1\sim p_0}\Big[V_1^{e\pi}(s_1) - V_1(s_1)\Big] = \sum_{h=1}^{H}\mathbb{E}_{s\sim\bar{d}_h^{e\pi}}\Big[\langle Q_h(s,\cdot),\, e\pi_h(\cdot\mid s) - \pi(\cdot\mid s)\rangle\Big] + \sum_{h=1}^{H}\mathbb{E}_{(s,a)\sim d_h^{e\pi}}\Big[e_h(s,a)\Big].$$

Since by definition

$$V_h(s) = \langle Q_h(s,\cdot),\, \pi(\cdot\mid s)\rangle,$$

we have

$$\langle Q_h(s,\cdot),\, e\pi_h(\cdot\mid s) - \pi(\cdot\mid s)\rangle = \langle Q_h(s,\cdot) - V_h(s),\, e\pi_h(\cdot\mid s) - \pi(\cdot\mid s)\rangle,$$

but the stated form is equivalent.

**(b)** For any $t \geq 1$ and stage $h$, define the model error at $(s,a)$ by

$$e_h^t(s,a) = R_h(s,a) + \big(P_h V_{h+1}^t\big)(s,a) - Q_h^t(s,a),$$

where $\{Q_h^t, V_h^t\}$ are computed on the estimated MDP (with estimated transitions $\widehat{P}_h^t$) and

$$V_h^t(s) = \langle Q_h^t(s,\cdot),\, \pi^t(\cdot|s)\rangle_A.$$

We assume that the transition model is estimated via a generative model so that for all $s, a, h$

$$\|P_h(\cdot\mid s,a) - \widehat{P}_h^t(\cdot\mid s,a)\|_1 \leq \frac{\sqrt{|\mathcal{S}|}}{t}.$$

Since the reward function is bounded in $[0,1]$ and the horizon is $H$, the *true* $Q$-functions are bounded by $H$ in absolute value. A standard error propagation argument (see, e.g., simulation

lemma type arguments) then implies that the error in the value function is at most proportional to the error in the model times the horizon. More precisely, one may show by induction that

$$\sup_{s,a,h} |e_h^t(s,a)| \leq H \cdot \frac{\sqrt{\mathcal{S}|}}{t},$$

for all $t \geq 1$.

The error $e_h^t(s,a)$ arises solely from the discrepancy between $P_h$ and $\widehat{P}_h^t$ when computing the one-step lookahead of $V_{h+1}^t$. Since the total variation error is bounded by $\frac{\sqrt{|\mathcal{S}|}}{t}$ and the value functions are bounded by $H$, the error in the expected value is at most $H \cdot \frac{p\,|\mathcal{S}|}{t}$.

**(c)** Let $V^\pi$ denote the state-value function of any policy $\pi$ on the true MDP and define

$$J(\pi) = \mathbb{E}_{s_1 \sim p_0}\big[V_1^\pi(s_1)\big].$$

Let $\pi^*$ be the optimal policy on the true MDP. Applying part (a) with $\pi = e\pi = \pi^*$ and the approximate value functions $\{Q_h^t, V_h^t\}$ obtained at iteration $t$, we get

$$J(\pi^*) - J(\pi_t) = \sum_{h=1}^{H} \mathbb{E}_{s\sim \bar{d}_h^{\pi^*}}\Big[\Big\langle Q_h^t(s,\cdot),\, \pi_h^*(\cdot \mid s) - \pi_t(\cdot \mid s)\Big\rangle_A\Big] + \sum_{h=1}^{H} \mathbb{E}_{(s,a)\sim d_h^{\pi^*}}\Big[e_h^t(s,a)\Big].$$

Using the bound from part (b), namely

$$\sup_{s,a,h} |e_h^t(s,a)| \leq H \cdot \frac{\sqrt{|\mathcal{S}}}{t},$$

and noting that the occupancy measure is a probability distribution at each stage (so that the expectation of the error is also bounded by $H \cdot \frac{\sqrt{|\mathcal{S}|}}{t}$ for each $h$), we obtain

$$\sum_{h=1}^{H} \mathbb{E}_{(s,a)\sim d_h^{\pi^*}}\Big[e_h^t(s,a)\Big] \leq \frac{H^2\sqrt{|\mathcal{S}|}}{t}.$$

In many analyses one incurs a factor of 2 to account for potential overestimation and underestimation, so that one can write

$$J(\pi^*) - J(\pi_t) \leq \frac{2H^2\sqrt{|\mathcal{S}|}}{t} + \sum_{h=1}^{H} \mathbb{E}_{s\sim d_h^{\pi^*}}\Big[\Big\langle Q_h^t(s,\cdot),\, \pi_h^*(\cdot \mid s) - \pi_t(\cdot \mid s)\Big\rangle_A\Big].$$

**(d)** Assume that the policy is updated via a Policy Mirror Descent (PMD) step with KL regularization:

$$\pi_h^{t+1} = \arg\max_{\pi_h \in \Delta(A)} \left\{\big\langle \alpha\, Q_h^t(s,\cdot),\, \pi_h - \pi_h^t\big\rangle - D_{\mathrm{KL}}(\pi_h \parallel \pi_h^t)\right\},$$

for each state $s$ and stage $h$. A standard regret bound for mirror descent (see, e.g., Beck and Teboulle) implies that for each fixed state $s$ and stage $h$, the cumulative linearized regret satisfies

$$\sum_{t=1}^{T}\Big\langle \alpha\, Q_h^t(s,\cdot),\, \pi_h^*(\cdot \mid s) - \pi_h^t(\cdot \mid s)\Big\rangle \le \frac{1}{\alpha} D_{\mathrm{KL}}(\pi_h^*(\cdot \mid s) \parallel \pi_h^1(\cdot \mid s)) + \alpha T\, G^2,$$

where $G = \max_{t,s} \|Q_h^t(s,\cdot)\|_\infty$ and the initialization $\pi^1$ is typically chosen as the uniform distribution so that

$$D_{\mathrm{KL}}(\pi_h^*(\cdot \mid s) \parallel \pi_h^1(\cdot \mid s)) \le \log|A|.$$

Choosing the step-size $\alpha$ to balance the two terms (typically $\alpha \propto 1/\sqrt{T}$), we obtain a per-state, per-stage regret of order $O\big(\sqrt{T}\,\big(\log|A|\big)\big)$. Integrating over all states $s$ (with weights given by $d_h^{\pi^*}(s)$) and summing over $h = 1, \ldots, H$ yields

$$\sum_{t=1}^{T}\sum_{h=1}^{H} \mathbb{E}_{s\sim d_h^{\pi^*}}\Big[\Big\langle Q_h^t(s,\cdot),\, \pi_h^*(\cdot \mid s) - \pi_h^t(\cdot \mid s)\Big\rangle_A\Big] \le C_1\, H\, \big(\sqrt{\log|A|}\big)\sqrt{T},$$

for some constant $C_1 > 0$.

Adding the error term from part (c) (which contributes a cumulative error of $\sum_{t=1}^{T} \frac{2H^2\sqrt{|\mathcal{S}|}}{t}$, and noting that

$$\sum_{t=1}^{T} \frac{1}{\sqrt{t}} \approx 2\sqrt{T}),$$

we obtain

$$\sum_{t=1}^{T}\Big[J(\pi^*) - J(\pi_t)\Big] \le C \cdot H^2 \cdot \Big(\sqrt{|\mathcal{S}|} + p\,\log|A|\Big)\sqrt{T},$$

for some constant $C > 0$.

# Problem Set 2 -- Problem 6

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import gymnasium as gym
import random
import torch

def set_all_seeds(seed):
    """Set all seeds for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)


def softmax(x, axis=-1):
    exp_x = np.exp(x - np.max(x, axis=axis, keepdims=True))
    return exp_x / np.sum(exp_x, axis=axis, keepdims=True)

def get_policy_from_theta(theta):
    return softmax(theta, axis=1)


def get_greedy_policy(policy, desc=None):
    """Extract the most probable actions from a policy.
    Size is automatically determined from the policy shape."""
    size = int(np.sqrt(policy.shape[0]))  # Calculate grid size from policy
    greedy_actions = np.argmax(policy, axis=1)

    print("\nMost probable actions and their probabilities:")
    for i in range(size):
        for j in range(size):
            state = i * size + j
            action = greedy_actions[state]
            prob = policy[state, action]
            print(f"[{action}:{prob:.2f}]", end=" ")
        print()

    print("\nOptimal Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):")
    print(greedy_actions.reshape(size, size))

    return greedy_actions

def plot_policy_grid(policy, desc):
    """Visualize policy with automatic size detection."""
```

```
    size = int(np.sqrt(len(policy)))  # Calculate grid size from policy leng

    # Create figure and axis
    fig, ax = plt.subplots(figsize=(12, 12))

    # Draw grid
    for i in range(size+1):
        ax.axhline(i, color='black', linewidth=1)
        ax.axvline(i, color='black', linewidth=1)

    # Set up the plot
    ax.set_xticks([])
    ax.set_yticks([])
    ax.invert_yaxis()

    # Dictionary for arrow directions
    arrows = {0: '←', 1: '↓', 2: '→', 3: '↑'}
    colors = {'S': 'green', 'F': 'lightblue', 'H': 'red', 'G': 'gold'}

    # Fill cells and add arrows
    for i in range(size):
        for j in range(size):
            state = i * size + j
            cell_type = desc[i][j].decode('utf-8')

            # Fill cell color based on type
            rect = plt.Rectangle((j, i), 1, 1, facecolor=colors[cell_type],
            ax.add_patch(rect)

            # Add cell type
            ax.text(j+0.1, i+0.2, cell_type, fontsize=12)

            # Add arrow for policy (except in holes and goal)
            if cell_type not in ['H', 'G']:
                action = int(policy[state])
                ax.text(j+0.5, i+0.5, arrows[action], fontsize=20, ha='cente

    plt.title('Policy Visualization\n(S: Start, F: Frozen, H: Hole, G: Goal)
    plt.tight_layout()
    return fig
```

In [2]:

```
def compute_true_Q(env, policy, gamma=0.99, tol=1e-6):
    """Compute true Q^π using the environment model."""
    n_states = env.observation_space.n
    n_actions = env.action_space.n

    Q_pi = np.zeros((n_states, n_actions))

    while True:
```

```
        Q_old = Q_pi.copy()

        for s in range(n_states):
            for a in range(n_actions):
                expected_value = 0
                for prob, next_state, reward, done in env.unwrapped.P[s][a]:
                    next_state_value = 0
                    for next_action in range(n_actions):
                        next_state_value += policy[next_state, next_action]
                    expected_value += prob * (reward + gamma * (1-done) * ne
                Q_pi[s, a] = expected_value

        if np.max(np.abs(Q_pi - Q_old)) < tol:
            break

    return Q_pi

def compute_optimal_Q(env, gamma=0.99, tol=1e-6):
    """Compute optimal Q* using value iteration."""
    n_states = env.observation_space.n
    n_actions = env.action_space.n

    Q = np.zeros((n_states, n_actions))

    while True:
        Q_old = Q.copy()

        for s in range(n_states):
            for a in range(n_actions):
                expected_value = 0
                for prob, next_state, reward, done in env.unwrapped.P[s][a]:
                    next_state_value = np.max(Q_old[next_state]) if not done
                    expected_value += prob * (reward + gamma * (1-done) * ne
                Q[s, a] = expected_value

        if np.max(np.abs(Q - Q_old)) < tol:
            break

    V = np.max(Q, axis=1)
    policy = np.zeros((n_states, n_actions))
    policy[range(n_states), np.argmax(Q, axis=1)] = 1

    return Q, V, policy
```

# Policy Evaluation -- This is the part you are going to write

You need to implement the `fit` function in the `QEstimator` class that performs

iterative policy evaluation using collected samples.

We implement **sample-based value iteration** for policy evaluation within policy iteration. The key idea is to iteratively refine the Q-function estimates using empirical averages from sampled trajectories.

## Data Collection

First, we generate a dataset of transitions by following the current policy ( \pi ). Each sample consists of:

$$(s, a, r, s', d)$$

where ( d ) is an indicator variable that equals ( 1 ) if the episode ends at ( s' ).

---

## Iterative Q-Function Updates

Since we do not have access to the true transition dynamics, we approximate the Q-values iteratively. The Q-function is represented as a table ( Q(s, a) ), initialized to zero. For each iteration, we update ( Q(s, a) ) using empirical estimates:

1. **Compute the State Value Function Approximation ( V_{\text{old}}(s') )**
   Since we only have samples, we approximate the value function of the next state by taking the expectation under the current policy:

   $$V_{\text{old}}(s') = \sum_{a'} \pi(a' \mid s') Q_{\text{old}}(s', a').$$

2. **Compute the Target for Each Sampled ( (s, a) ) Pair**
   Using the Bellman equation, we define the target value:

   $$\text{Target}(s, a) = r + \gamma(1 - d) V_{\text{old}}(s').$$

   The term ( (1 - d) ) ensures that no future rewards are counted if the episode terminates.

3. **Compute the Empirical Average for Each ( (s, a) ) Pair**
   The update is performed by averaging over all observed transitions for a given ( (s, a) ):

   $$Q_{\text{new}}(s, a) = \frac{\sum \text{Target}(s, a)}{\sum \text{Count}(s, a)},$$

where $\sum \text{Target}(s, a)$ is the accumulated sum of target values and $\sum \text{Count}(s, a)$ is the number of times ( (s, a) ) was sampled.

4. **Continue Until Convergence**
   This process continues until the maximum change in Q-values falls below a threshold ( \epsilon ).

---

The above logic is what you are going to implement.

In [3]:
```python
class QEstimator:
    def __init__(self, n_states, n_actions):
        self.n_states = n_states
        self.n_actions = n_actions
        self.w = np.zeros((n_states, n_actions))

    def fit(self, samples, gamma, policy, n_iterations=100, tol=1e-2):
        """Policy evaluation using samples to estimate Q^π."""
        states = np.array([s for s, _, _, _, _ in samples])
        actions = np.array([a for _, a, _, _, _ in samples])
        rewards = np.array([r for _, _, r, _, _ in samples])
        next_states = np.array([s_next for _, _, _, s_next, _ in samples])
        dones = np.array([d for _, _, _, _, d in samples])

        # Create arrays to store targets for each (s,a) pair
        counts = np.zeros((self.n_states, self.n_actions))
        targets_sum = np.zeros((self.n_states, self.n_actions))
        max_diff_history = []

        for iter_idx in range(n_iterations):
            old_w = self.w.copy()

            # Reset sums for this iteration
            counts.fill(0)
            targets_sum.fill(0)

            ## YOUR CODE STARTS FROM HERE

            # 1. Compute the estimated value function for next states
            #    - Use the current Q-table `self.w`
            #    - Weight by policy probabilities π(a' | s')

            v_next = np.zeros(len(next_states))
            for i, s_next in enumerate(next_states):
                pi_probs = policy[s_next]
                v_next[i] = np.dot(pi_probs, self.w[s_next])

            # 2. Compute the target using the Bellman equation
```

```
            #    - Use sampled rewards and estimated next-state value
            #    - Zero out next-state value if `done` is True

            targets = rewards + gamma * (1 - dones) * v_next

            # 3. Accumulate target values for each state-action pair
            #    - Sum up target values and count occurrences for averaging

            np.add.at(targets_sum, (states, actions), targets)
            np.add.at(counts, (states, actions), 1)

            # 4. Compute the new Q-value estimate using sample averages
            #    - Only update Q-values where at least one sample was observ

            mask = counts > 0
            self.w[mask] = targets_sum[mask] / counts[mask]

            ## YOUR CODE ENDS HERE


            # Check convergence
            max_diff = np.max(np.abs(self.w - old_w))
            max_diff_history.append(max_diff)

            if max_diff < tol:
                # print(f"Policy evaluation converged after {iter_idx + 1} i
                return True

        print(f"Warning: Policy evaluation did not converge after {n_iterati
        print(f"Max diff history: {', '.join([f'{d:.6f}' for d in max_diff_h
        return False
```

# Conservative Policy Iteration: Frank-Wolfe + Policy Evaluation

In [4]:
```
def frank_wolfe_with_samples(env, n_iterations=500, episodes_per_iter=500,
                             max_steps=100, gamma=0.99):
    n_states = env.observation_space.n
    n_actions = env.action_space.n

    policy = np.ones((n_states, n_actions)) / n_actions
    performance_history = []
    pe_error_history = []

    q_estimator = QEstimator(n_states, n_actions)

    for t in range(n_iterations):
        # Collect samples
```

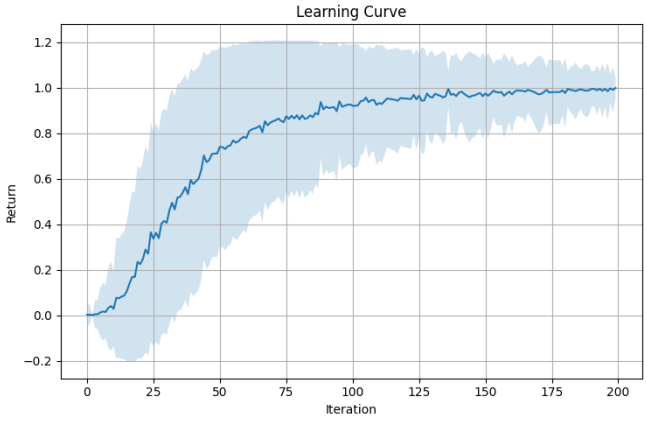
```
        all_samples = []
        episode_returns = []

        for _ in range(episodes_per_iter):
            state, _ = env.reset()
            episode_return = 0
            done = False
            step = 0

            while not done and step < max_steps:
                action = np.random.choice(n_actions, p=policy[state])
                next_state, reward, terminated, truncated, _ = env.step(acti
                done = terminated or truncated
                all_samples.append((state, action, reward, next_state, done)
                episode_return += reward
                state = next_state
                step += 1

            episode_returns.append(episode_return)

        avg_return = np.mean(episode_returns)
        std_return = np.std(episode_returns)
        performance_history.append((avg_return, std_return))

        # Policy evaluation
        converged = q_estimator.fit(all_samples, gamma, policy)

        # Compare with true Q-values for PE error
        Q_pi_true = compute_true_Q(env, policy, gamma)
        pe_error = np.max(np.abs(q_estimator.w - Q_pi_true))
        pe_error_history.append(pe_error)

        if t % 40 == 0:
            print(f"Iteration {t}:")
            print(f"Return = {avg_return:.4f} ± {std_return:.4f}")
            print(f"Policy Evaluation Error = {pe_error:.6f}\n")

        # Frank-Wolfe update
        eta = 0.02

        for s in range(n_states):
            best_action = np.argmax(q_estimator.w[s])
            s_pi = np.zeros(n_actions)
            s_pi[best_action] = 1.0
            policy[s] = (1 - eta) * policy[s] + eta * s_pi

    # Evaluate final learned policy
    final_returns = []
    for _ in range(20):
        state, _ = env.reset()
```

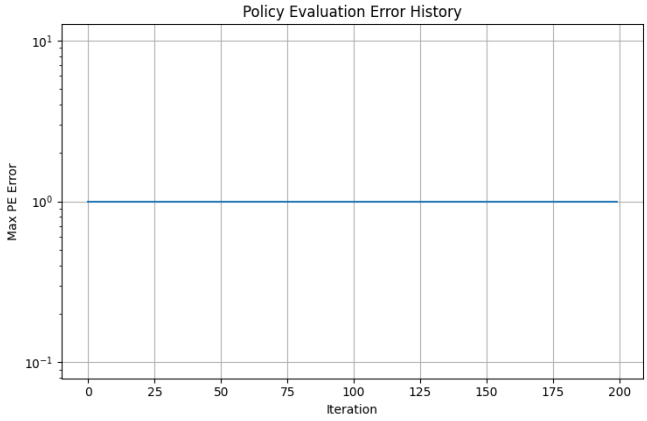
```
        episode_return = 0
        done = False
        step = 0
        while not done and step < max_steps:
            action = np.random.choice(n_actions, p=policy[state])
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode_return += reward
            state = next_state
            step += 1
        final_returns.append(episode_return)

    # Compute optimal policy and evaluate it
    _, _, pi_star = compute_optimal_Q(env, gamma)
    optimal_returns = []
    for _ in range(20):
        state, _ = env.reset()
        episode_return = 0
        done = False
        step = 0
        while not done and step < max_steps:
            action = np.random.choice(n_actions, p=pi_star[state])
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode_return += reward
            state = next_state
            step += 1
        optimal_returns.append(episode_return)

    print(f"\nFinal Policy Average Return (20 episodes): {np.mean(final_retu
    print(f"Optimal Policy Average Return (20 episodes): {np.mean(optimal_re

    # Plot results
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    # Plot returns
    returns = [p[0] for p in performance_history]
    stds = [p[1] for p in performance_history]
    ax1.plot(returns)
    ax1.fill_between(range(len(returns)),
                     np.array(returns) - np.array(stds),
                     np.array(returns) + np.array(stds),
                     alpha=0.2)
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Return')
    ax1.set_title('Learning Curve')
    ax1.grid(True)

    # Plot policy evaluation error
    ax2.plot(pe_error_history)
```

```
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Max PE Error')
    ax2.set_title('Policy Evaluation Error History')
    ax2.set_yscale('log')
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

    return policy, pi_star
```

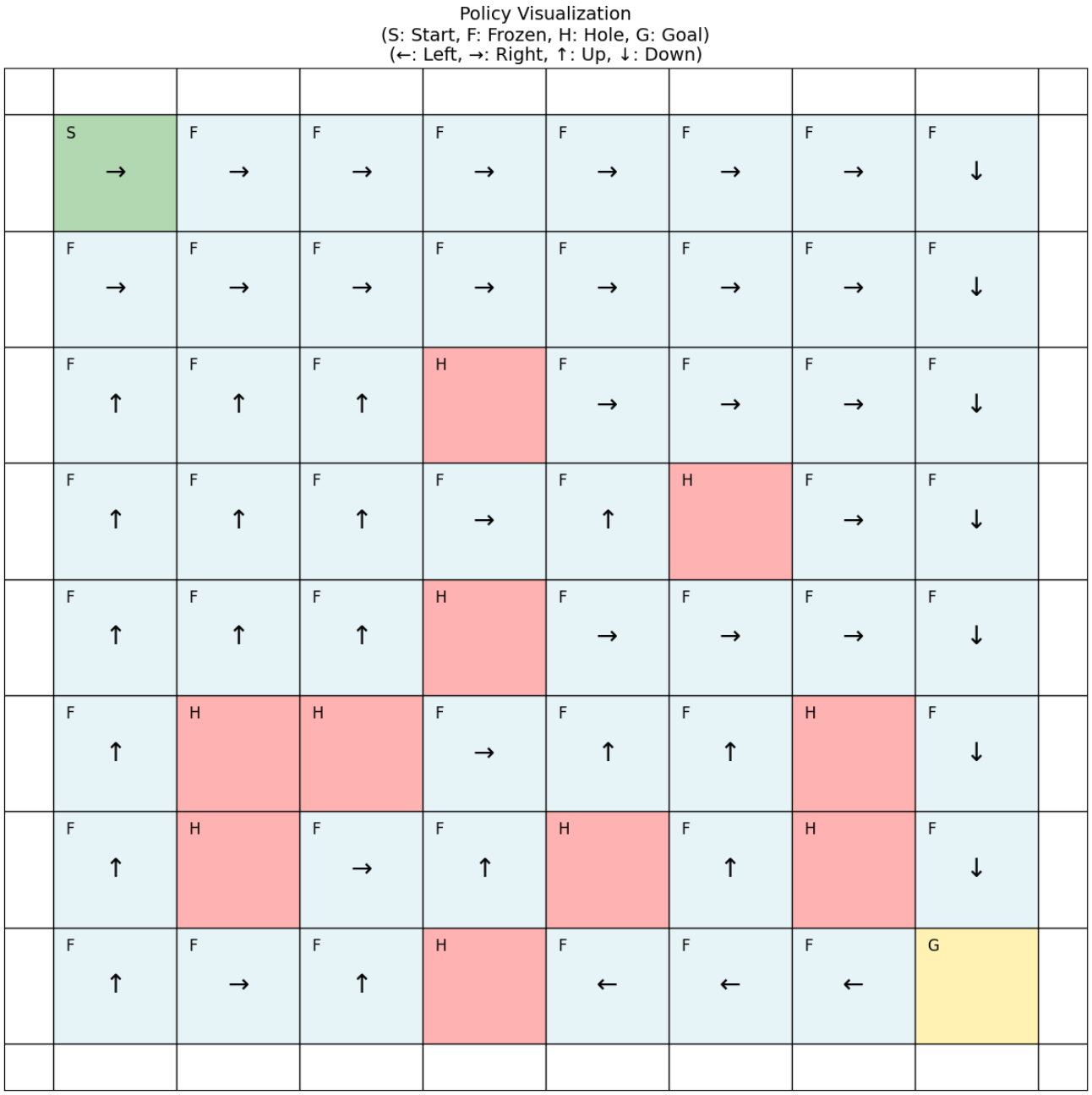
In [5]:
```
np.random.seed(24)
env = gym.make('FrozenLake-v1', map_name="8x8", is_slippery=False)

final_policy, pi_star = frank_wolfe_with_samples(
    env,
    n_iterations=200,
    episodes_per_iter=500,
    max_steps=100,
    gamma=0.99
)

# Visualize policies
print("\nLearned Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):")
pi_greedy = get_greedy_policy(final_policy)
plot_policy_grid(pi_greedy, env.unwrapped.desc)
plt.show()
```

```
Iteration 0:
Return = 0.0020 ± 0.0447
Policy Evaluation Error = 1.000000

Iteration 40:
Return = 0.5760 ± 0.4942
Policy Evaluation Error = 1.000000

Iteration 80:
Return = 0.8600 ± 0.3470
Policy Evaluation Error = 1.000000

Iteration 120:
Return = 0.9520 ± 0.2138
Policy Evaluation Error = 1.000000

Iteration 160:
Return = 0.9700 ± 0.1706
Policy Evaluation Error = 1.000000


Final Policy Average Return (20 episodes): 1.0000 ± 0.0000
Optimal Policy Average Return (20 episodes): 1.0000 ± 0.0000
```



```
Learned Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):

Most probable actions and their probabilities:
[2:0.99] [2:0.99] [2:0.99] [2:0.99] [2:0.99] [2:0.99] [2:0.98] [1:0.99]
[2:0.60] [2:0.73] [2:0.80] [2:0.99] [2:0.99] [2:0.99] [2:0.98] [1:0.99]
[3:0.98] [3:0.98] [3:0.98] [0:0.99] [2:0.69] [2:0.99] [2:0.98] [1:0.99]
[3:0.98] [3:0.98] [3:0.98] [2:0.99] [3:0.97] [0:0.99] [2:0.99] [1:0.99]
[3:0.98] [3:0.99] [3:0.99] [0:0.99] [2:0.99] [2:0.99] [2:0.99] [1:0.99]
[3:0.99] [0:0.99] [0:0.99] [2:0.99] [3:0.99] [3:0.99] [0:0.99] [1:0.99]
[3:0.99] [0:0.99] [2:0.99] [3:0.99] [0:0.99] [3:0.98] [0:0.99] [1:0.99]
[3:0.98] [2:0.99] [3:0.99] [0:0.99] [0:0.99] [0:0.99] [0:0.99] [0:0.99]

Optimal Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):
[[2 2 2 2 2 2 2 1]
 [2 2 2 2 2 2 2 1]
 [3 3 3 0 2 2 2 1]
 [3 3 3 2 3 0 2 1]
 [3 3 3 0 2 2 2 1]
 [3 0 0 2 3 3 0 1]
 [3 0 2 3 0 3 0 1]
 [3 2 3 0 0 0 0 0]]
```




# Policy Mirror Descent: Mirror Descent + Policy Evaluation

In [6]:
```
def policy_mirror_descent_with_samples(env, n_iterations=500, episodes_per_i
                                       max_steps=100, alpha=0.1, gamma=0.99):
    n_states = env.observation_space.n
    n_actions = env.action_space.n

    theta = np.zeros((n_states, n_actions))
    performance_history = []
    pe_error_history = []
```

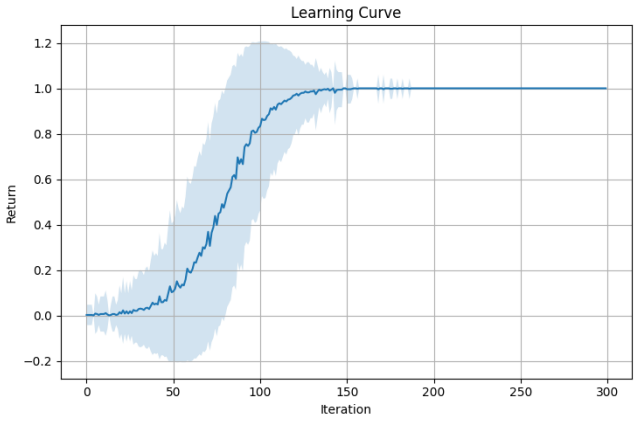
```
    q_estimator = QEstimator(n_states, n_actions)

    for t in range(n_iterations):
        policy = get_policy_from_theta(theta)

        # Collect samples
        all_samples = []
        episode_returns = []

        for _ in range(episodes_per_iter):
            state, _ = env.reset()
            episode_return = 0
            done = False
            step = 0

            while not done and step < max_steps:
                action = np.random.choice(n_actions, p=policy[state])
                next_state, reward, terminated, truncated, _ = env.step(acti
                done = terminated or truncated
                all_samples.append((state, action, reward, next_state, done)
                episode_return += reward
                state = next_state
                step += 1

            episode_returns.append(episode_return)

        avg_return = np.mean(episode_returns)
        std_return = np.std(episode_returns)
        performance_history.append((avg_return, std_return))

        # Policy evaluation
        converged = q_estimator.fit(all_samples, gamma, policy)

        # Compare with true Q-values for PE error
        Q_pi_true = compute_true_Q(env, policy, gamma)
        pe_error = np.max(np.abs(q_estimator.w - Q_pi_true))
        pe_error_history.append(pe_error)

        if t % 40 == 0:
            print(f"Iteration {t}:")
            print(f"Return = {avg_return:.4f} ± {std_return:.4f}")
            print(f"Policy Evaluation Error = {pe_error:.6f}\n")

        # Mirror descent update
        theta += alpha * q_estimator.w

    final_policy = get_policy_from_theta(theta)

    # Evaluate final learned policy
```

```
final_returns = []
for _ in range(20):
    state, _ = env.reset()
    episode_return = 0
    done = False
    step = 0
    while not done and step < max_steps:
        action = np.random.choice(n_actions, p=final_policy[state])
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        episode_return += reward
        state = next_state
        step += 1
    final_returns.append(episode_return)

# Compute optimal policy and evaluate it
_, _, pi_star = compute_optimal_Q(env, gamma)
optimal_returns = []
for _ in range(20):
    state, _ = env.reset()
    episode_return = 0
    done = False
    step = 0
    while not done and step < max_steps:
        action = np.random.choice(n_actions, p=pi_star[state])
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        episode_return += reward
        state = next_state
        step += 1
    optimal_returns.append(episode_return)

print(f"\nFinal Policy Average Return (20 episodes): {np.mean(final_retu
print(f"Optimal Policy Average Return (20 episodes): {np.mean(optimal_re

# Plot results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

# Plot returns
returns = [p[0] for p in performance_history]
stds = [p[1] for p in performance_history]
ax1.plot(returns)
ax1.fill_between(range(len(returns)),
                 np.array(returns) - np.array(stds),
                 np.array(returns) + np.array(stds),
                 alpha=0.2)
ax1.set_xlabel('Iteration')
ax1.set_ylabel('Return')
ax1.set_title('Learning Curve')
ax1.grid(True)
```

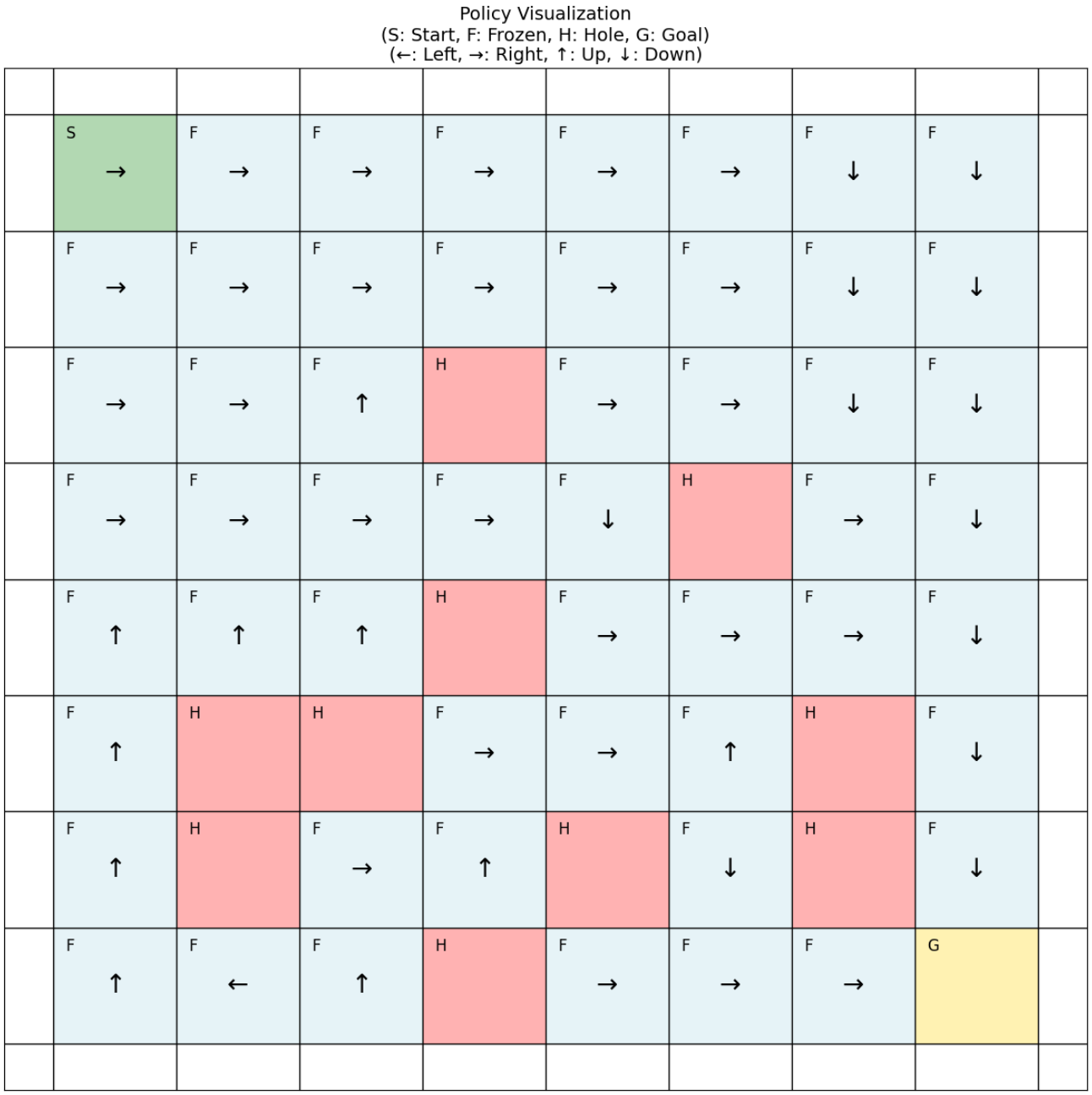
```
    # Plot policy evaluation error
    ax2.plot(pe_error_history)
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Max PE Error')
    ax2.set_title('Policy Evaluation Error History')
    ax2.set_yscale('log')
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

    return final_policy, pi_star
```

In [7]:
```
# Set random seed
np.random.seed(24)
env = gym.make('FrozenLake-v1', map_name="8x8", is_slippery=False)

final_policy, pi_star = policy_mirror_descent_with_samples(
    env,
    n_iterations=300,
    episodes_per_iter=500,
    max_steps=100,
    alpha=0.1,
    gamma=0.99
)

# Visualize policies
print("\nLearned Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):")
pi_greedy = get_greedy_policy(final_policy)
plot_policy_grid(pi_greedy, env.unwrapped.desc)
plt.show()
```

```
Iteration 0:
Return = 0.0020 ± 0.0447
Policy Evaluation Error = 1.000000

Iteration 40:
Return = 0.0520 ± 0.2220
Policy Evaluation Error = 0.677858

Iteration 80:
Return = 0.5020 ± 0.5000
Policy Evaluation Error = 0.821186

Iteration 120:
Return = 0.9700 ± 0.1706
Policy Evaluation Error = 0.924676

Iteration 160:
Return = 1.0000 ± 0.0000
Policy Evaluation Error = 0.963327

Iteration 200:
Return = 1.0000 ± 0.0000
Policy Evaluation Error = 0.967393

Iteration 240:
Return = 1.0000 ± 0.0000
Policy Evaluation Error = 0.967928

Iteration 280:
Return = 1.0000 ± 0.0000
Policy Evaluation Error = 0.968095


Final Policy Average Return (20 episodes): 1.0000 ± 0.0000
Optimal Policy Average Return (20 episodes): 1.0000 ± 0.0000
```




```
Learned Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):

Most probable actions and their probabilities:
[2:0.41] [2:0.52] [2:0.54] [2:0.52] [2:0.45] [2:0.39] [1:0.42] [1:0.48]
[2:0.40] [2:0.53] [2:0.61] [2:0.73] [2:0.54] [2:0.50] [1:0.51] [1:0.61]
[2:0.35] [2:0.34] [3:0.50] [0:0.25] [2:0.65] [2:0.82] [1:0.49] [1:0.66]
[2:0.44] [2:0.62] [2:0.78] [2:0.95] [1:0.76] [0:0.25] [2:0.52] [1:0.69]
[3:0.40] [3:0.51] [3:0.89] [0:0.25] [2:0.85] [2:0.89] [2:0.87] [1:0.72]
[3:0.82] [0:0.25] [0:0.25] [2:1.00] [2:0.58] [3:0.55] [0:0.25] [1:0.77]
[3:1.00] [0:0.25] [2:0.45] [3:0.99] [0:0.25] [1:0.93] [0:0.25] [1:0.78]
[3:0.94] [0:0.44] [3:0.28] [0:0.25] [2:1.00] [2:0.91] [2:0.93] [0:0.25]

Optimal Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):
[[2 2 2 2 2 2 1 1]
 [2 2 2 2 2 2 1 1]
 [2 2 3 0 2 2 1 1]
 [2 2 2 2 1 0 2 1]
 [3 3 3 0 2 2 2 1]
 [3 0 0 2 2 3 0 1]
 [3 0 2 3 0 1 0 1]
 [3 0 3 0 2 2 2 0]]
```




# Policy Iteration

In [8]:
```
def policy_iteration_with_samples(env, n_iterations=500, episodes_per_iter=5
                                  max_steps=100, gamma=0.99):
    n_states = env.observation_space.n
    n_actions = env.action_space.n

    policy = np.ones((n_states, n_actions)) / n_actions
    performance_history = []
    pe_error_history = []
```

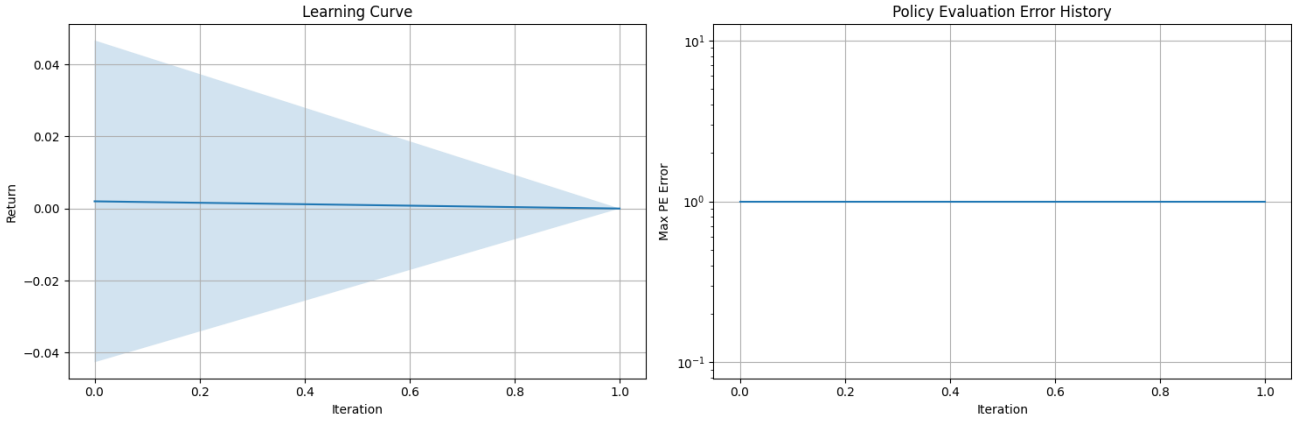
```
    q_estimator = QEstimator(n_states, n_actions)

    for t in range(n_iterations):
        # Collect samples
        all_samples = []
        episode_returns = []

        for _ in range(episodes_per_iter):
            state, _ = env.reset()
            episode_return = 0
            done = False
            step = 0

            while not done and step < max_steps:
                action = np.random.choice(n_actions, p=policy[state])
                next_state, reward, terminated, truncated, _ = env.step(acti
                done = terminated or truncated
                all_samples.append((state, action, reward, next_state, done)
                episode_return += reward
                state = next_state
                step += 1

            episode_returns.append(episode_return)

        avg_return = np.mean(episode_returns)
        std_return = np.std(episode_returns)
        performance_history.append((avg_return, std_return))

        # Policy evaluation
        converged = q_estimator.fit(all_samples, gamma, policy)

        # Compare with true Q-values for PE error
        Q_pi_true = compute_true_Q(env, policy, gamma)
        pe_error = np.max(np.abs(q_estimator.w - Q_pi_true))
        pe_error_history.append(pe_error)

        if t % 40 == 0:
            print(f"Iteration {t}:")
            print(f"Return = {avg_return:.4f} ± {std_return:.4f}")
            print(f"Policy Evaluation Error = {pe_error:.6f}\n")


        # Policy improvement (greedy update)
        new_policy = np.zeros_like(policy)
        new_policy[range(n_states), np.argmax(q_estimator.w, axis=1)] = 1

        # Check if policy has changed
        if np.array_equal(policy, new_policy):
            print(f"Policy converged at iteration {t}")
            break
```

```
        policy = new_policy

    # Evaluate final learned policy
    final_returns = []
    for _ in range(20):
        state, _ = env.reset()
        episode_return = 0
        done = False
        step = 0
        while not done and step < max_steps:
            action = np.random.choice(n_actions, p=policy[state])
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode_return += reward
            state = next_state
            step += 1
        final_returns.append(episode_return)

    # Compute optimal policy and evaluate it
    _, _, pi_star = compute_optimal_Q(env, gamma)
    optimal_returns = []
    for _ in range(20):
        state, _ = env.reset()
        episode_return = 0
        done = False
        step = 0
        while not done and step < max_steps:
            action = np.random.choice(n_actions, p=pi_star[state])
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            episode_return += reward
            state = next_state
            step += 1
        optimal_returns.append(episode_return)

    print(f"\nFinal Policy Average Return (20 episodes): {np.mean(final_retu
    print(f"Optimal Policy Average Return (20 episodes): {np.mean(optimal_re

    # Plot results
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))

    # Plot returns
    returns = [p[0] for p in performance_history]
    stds = [p[1] for p in performance_history]
    ax1.plot(returns)
    ax1.fill_between(range(len(returns)),
                     np.array(returns) - np.array(stds),
                     np.array(returns) + np.array(stds),
                     alpha=0.2)
```

```
    ax1.set_xlabel('Iteration')
    ax1.set_ylabel('Return')
    ax1.set_title('Learning Curve')
    ax1.grid(True)

    # Plot policy evaluation error
    ax2.plot(pe_error_history)
    ax2.set_xlabel('Iteration')
    ax2.set_ylabel('Max PE Error')
    ax2.set_title('Policy Evaluation Error History')
    ax2.set_yscale('log')
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

    return policy, pi_star
```

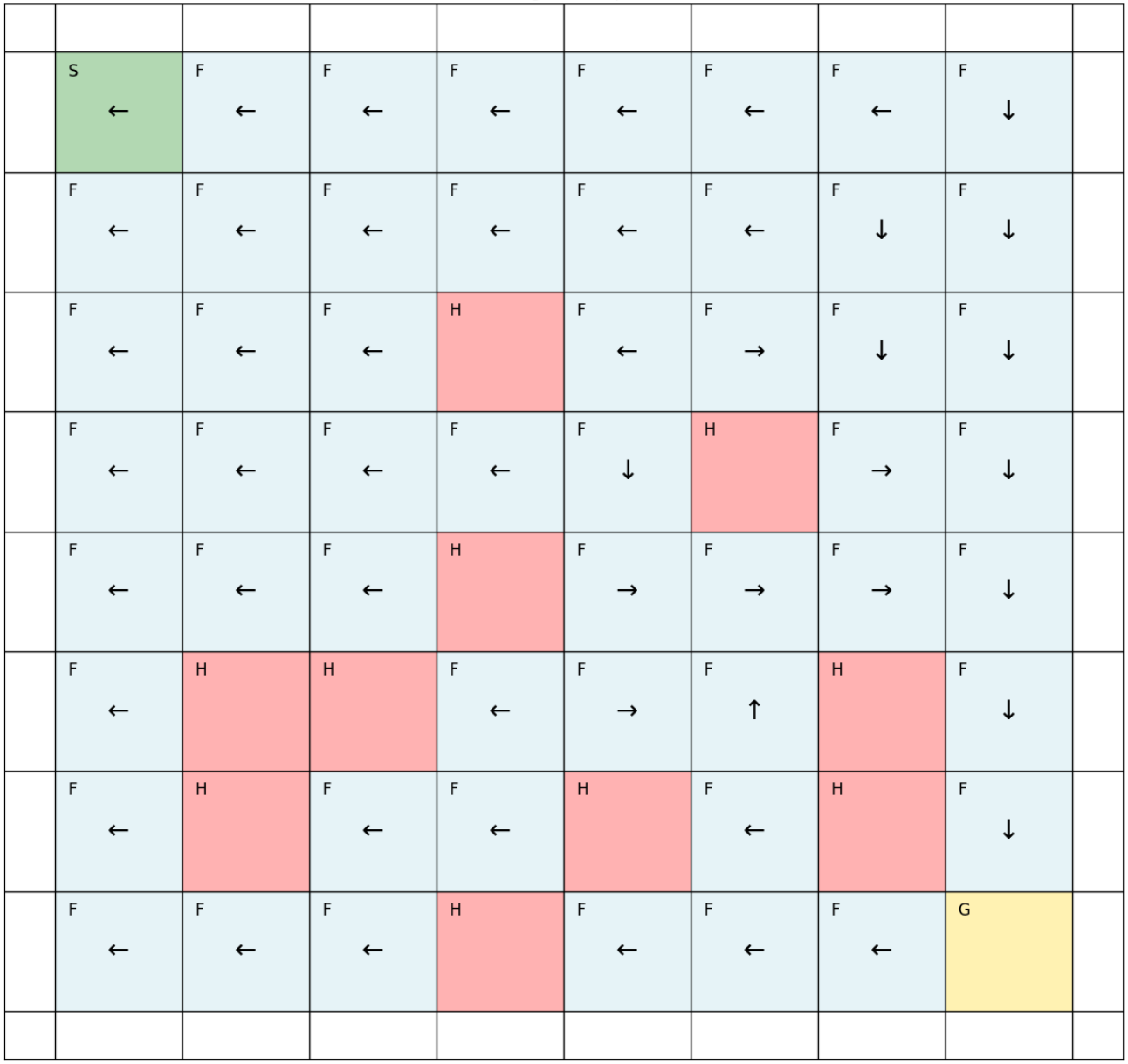
In [9]:
```
np.random.seed(24)
env = gym.make('FrozenLake-v1', map_name="8x8", is_slippery=False)

final_policy, pi_star = policy_iteration_with_samples(
    env,
    n_iterations=200,
    episodes_per_iter=500,
    max_steps=100,
    gamma=0.99
)

# Visualize policies
print("\nLearned Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):")
pi_greedy = get_greedy_policy(final_policy)
plot_policy_grid(pi_greedy, env.unwrapped.desc)
plt.show()
```

```
Iteration 0:
Return = 0.0020 ± 0.0447
Policy Evaluation Error = 1.000000

Policy converged at iteration 1

Final Policy Average Return (20 episodes): 0.0000 ± 0.0000
Optimal Policy Average Return (20 episodes): 1.0000 ± 0.0000
```




```
Learned Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):

Most probable actions and their probabilities:
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [1:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [2:1.00] [1:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [1:1.00] [0:1.00] [2:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [2:1.00] [2:1.00] [2:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [2:1.00] [3:1.00] [0:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [1:1.00]
[0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00] [0:1.00]

Optimal Policy (0:LEFT, 1:DOWN, 2:RIGHT, 3:UP):
[[0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 1 1]
 [0 0 0 0 0 2 1 1]
 [0 0 0 0 1 0 2 1]
 [0 0 0 0 2 2 2 1]
 [0 0 0 0 2 3 0 1]
 [0 0 0 0 0 0 0 1]
 [0 0 0 0 0 0 0 0]]
```

Policy Visualization
(S: Start, F: Frozen, H: Hole, G: Goal)
(←: Left, →: Right, ↑: Up, ↓: Down)




In [ ]: